GUILE: Graphical User Interface for Linguistic Experiments

Padmanabhan Krishnan[1]
Bruce McKenzie[2]
Stephen Hunt

Department of Computer Science
University of Canterbury, Private Bag 4800
Christchurch, New Zealand

Abstract

In this paper we explain a graphical user interface for defining semantic descriptions of programming languages. Our tool is implemented using SUIT and permits the development of action descriptions of programming languages.

1 Introduction

A programming language can be viewed as a specification language in which all specifications (programs) are executable. In order to use a programming language it is necessary to construct a system that can execute the programs. This is usually achieved using compilers or interpreters. Writing a compiler for a language requires the knowledge of the syntax for the language, the semantics of the various constructs in the language and the syntax and semantics of the target language. The structure of a typical compiler consists of a syntax analyser, a semantic analyser and a code generator. The construction of a compiler is simplified by using tools such as lexical analyser generators and parser generators. These tools are mainly used only for the syntactic analysis phase. The implementor specifies the semantic analysis and code generation as a program fragment. The limitation of such a technique has been well recognised [7]. The drawbacks can be attributed to the fact that semantic analysis and code generation are the most difficult issues and a connection to the intended semantics of the the language needs to be established. Compiler generation from formal semantics aims to simplify the construction of semantic analysers. A compiler generator takes as input the definition of a language and produces a compiler for a particular target. The task now is to construct a precise and complete definition (an explicit formal semantics) of the required language.

Traditionally a denotational semantics (i.e., using the λ-notation) of a programming language has been used to generate compilers [12]. Another approach has been to translate the λ-notation into other abstraction machines [5, 6]. The shortcomings of the λ-notation has been discussed in [7] where a new notation for defining the semantics of a programming language is developed. However, the notation is ad-hoc and hence the issue of correctness of the compilers being generated cannot be addressed. Action Notation [9] has been designed to provide a framework to describe the semantics of realistic programming languages. The feasibility of using the Action Notation to generate correct compilers has been shown [10, 11].

In the process of developing a programming language, the designer would normally wish to experiment with various constructs. This permits an evaluation of the constructs for issues such as effectiveness of the constructs and ease of use. To ease this process, [14] describes the need for a language designer's workbench. The two principal components of the workbench are a framework for defining semantics and a compiler generator which is based on the chosen framework. For the workbench to be accessible to a wide audience, the framework for defining semantics must be easy to understand, support rapid development of semantics and provide support for maintaining the semantics. Using such a workbench, the language designer would be able to experiment with a new language and encode the design decisions in the semantics. The compiler generator can then be used to obtain a compiler that can in turn be used to execute programs and evaluate the various language constructs.

[1] E-Mail:paddy@cosc.canterbury.ac.nz
[2] E-Mail:bruce@cosc.canterbury.ac.nz

One of the reasons why a language designer's workbench has not been used is the lack of an easy to understand specification language in which semantics for programming languages can be constructed. Notations for semantics are usually viewed as a forbidding collection of abstruse symbols. The selected symbols could have a more readable representation without complicating the semantics. For example, the Action Notation [9] achieves this where the choice of English words leads to better readability. What is more difficult is to get the practitioners to write semantics. This is because the semantics of the constructs have to be well understood and hence writing a formal description is harder than understanding an existing one. In other words, the semantics of the specification language is a major hurdle.

We contend that by designing an appropriate graphical interface, the user can be better persuaded to define the semantics of a programming language. Towards describing the specification language, we replace abstract formulae by suggestive pictures. The pictures help the user to focus on the main concept by ignoring the low level details. In this article we describe a graphical interface tool for developing Action descriptions of programming languages.

In the following sections we describe the details of our approach. In the next section we present a brief overview of the Action Notation and a sample semantic description. The main part of the paper, is presented in section 3 where the example in section 2 is developed using the tool. In section 4 our experience with the tool is summarised.

2 The Action Notation

The Action Notation is a high level notation used to construct useful descriptions of programming languages. The notation consists of primitive actions and combinators which are used to combine actions to define more complex actions. Actions are objects which can be performed to process information and are used to represent semantics of programs.

Primitive actions identify the elementary behaviour of information processing such as specification of control, generating and passing values and bindings, memory management and distributed processing. To ease the use of the notation the information processed by actions is divided into various facets which includes control, functional, declarative, imperative and communicative. The control facet is concerned with the flow of control, the functional facet is concerned with temporary intermediate values, the declarative facet is concerned with scoped information as in symbol tables, the imperative facet deals with stores corresponding to variables and their values and the communicative facet deals with messages between distributed units.

In general, a primitive action processes a single information facet while a combinator defines both the flow of control and the flow of information. In the remainder of this section we present a brief and informal summary of a subset of the notation. The reader is referred to [9] (pages 261-277) for details.

The control actions include complete, diverge, fail and escape. The action complete an always terminates, while diverge never terminates. The action fail indicates abortive termination and is used to abandon the current alternative. The action escape corresponds to raising an exception.

The control combinators include or, and, and then, trap and unfolding. The combinator or represents non-deterministic choice. An alternative to the chosen action is performed when the chosen action fails. The combinator and performs two actions with arbitrary interleaving while the combinator and then corresponds to sequential performance. The combinator trap is used to handle exceptions raised using escape. The combinator unfolding along with the basic action unfold specifies iteration. unfolding A performs A, but when unfold is encountered in A, the action A is performed.

The functional actions process intermediate values and give/are given data. The action give D yields the datum D while the action give D#n yields the n'th component

of the tuple represented by D. The action regive regenerates any data given to it and is useful to make copies of the given data. The action check D completes if D is the boolean true; fails otherwise. The principal functional combinator is then. A_1 then A_2 corresponds to functional composition, i.e., A_2 is given the data produced by A_1.

The declarative actions process scoped information and associate tokens (identifiers in the semantic domain) with values. The actions include bind T to D, which produces a binding of token T to datum D and rebind which reproduces all the bindings it received and produce D which converts the data item D into a binding. Information from the current bindings can be extracted by the S bound to T returns the datum (if it is of sort S) bound to the token T. The data specification current bindings converts the entire set of bindings into data. This combined with produce permits the manipulation of bindings as data and reconverting data into bindings.

The declarative combinators include moreover, furthermore, and hence. The action A_1 moreover A_2 corresponds to letting bindings produced by A_2 override those produced by A_1, i.e., bindings produced by A_2 have a higher precedence. The action furthermore A is similar and produces the same bindings as A along with any received bindings that is not overridden by A. The action A_1 hence A_2 restricts the bindings received by A_2 to those produced by A_1 and bindings produce by A_2 is propagated. This limits the scope of bindings produced by A1 unless A2 reproduces them.

The imperative actions deal with storage, consisting of individual cells, which is stable information. The actions include store and allocate. The action store D_1 in D_2 stores the datum D_1 in cell D_2 while allocate D corresponds to the allocation of a cell of sort D. Data of sort S that is stored in a cell D can be extracted by the S stored in D.

We now present a few examples which illustrate the use of the action notation in describing programming constructs.

The first equation we present defines the meaning of evaluating the addition of two expressions. If the order of expression evaluation is unspecified, we use the and combinator, i.e., arbitrary interleaving. If the expressions are to be evaluated left to right the and can be replaced by and then. On evaluating the two expressions, the two partial results are added. The use of number#1 refers to the data generated by evaluate E1 and requires that it be a number. The two partial results number#1 and number#2 is passed to the next action (using the then combinator) to be added. To evaluate a constant, the symbol table entry for the identifier is consulted.

```
evaluate ⟦ Expression1 "+" Expression2 ⟧ =
    | evaluate Expression1 and
    | evaluate Expression2
  then
    | give the sum(number#1, number#2)

evaluate Identifier = give the datum bound
    to token of Identifier
```

Those who are not experts in the notation tend to use and then instead of then. However given a pictorial representation where the passing of a value is made explicit, the user is influenced to make the right choice.

The semantics of a block (i.e., a set of declarations followed by a sequence of statements), is described by the equation execute. The first step is to update the symbol table according to the new declarations, which is specified by elaborate. The current bindings are extended by the declarations (i.e., current definitions hide old ones) using furthermore and then the statements are executed.

```
execute ⟦ Declaration "begin"
    Statement "end" ";" ⟧ =
    | furthermore elaborate Declaration
  hence
    | execute Statement
```

In the descriptions produced by beginners, the combinator furthermore is often missing. While they are aware that a block can use existing definitions (which are not over-ridden), the formal specification does not reflect it. Similarly, there is often confusion about the choice of combinator for

hence. We suspect that the two issues are related and the design of the right icons will help the user to construct the intended definition.

As an example of statement execution, we consider the while loop which evaluates the expression, and if it is true executes the body and then starts the next iteration.

```
execute ⟦ "while" Expression "loop" Statement "end" "loop" ";" ⟧
    unfolding
    | evaluate Expression then
    | | check (it is true) and then
    | | | execute Statement and then unfold
    | or
    | | check (it is false)
```

As an example of elaborate, the declaration of a variable results in an allocation of a cell for the value and binding the identifier to the cell.

```
elaborate ⟦ Identifier ":" "integer" ";" ⟧ =
    allocate (a number cell) then
    bind token of Integer to the cell#1
```

The semantic function execute can be extended to include the assignment statement. Towards the meaning of the assignment statement, the l-value of the identifier is obtained (access) and the r-value of the expression is computed. The cell indicated by the l-value is updated. To obtain the r-value of an identifier, the l-value is obtained and then r-value retrieved from the cell.

```
execute ⟦ Expression1 ":=" Expression2 ⟧ =
    | access Expression1 and
    | evaluate Expression2
    then
    | store (the given value#2) in
    | (the given variable#1)

access Identifier = give the cell bound to token of Identifier

evaluate Identifier = access Identifier then
    give the datum stored in it
```

The above definitions are purely for illustration. More detailed examples can be found in [9] where a complete semantics for a large subset of Ada [1] is presented.

3 Constructing Action Descriptions

Pictorial representation of complex structures is not a new concept, e.g, flow charts to represent algorithmic content of programs. Complicated data base structures have been represented using the entity-relationship model [15]. For example, an entity is represented as a box and a relationship is represented as a diamond. Attributes such as connectivity of the relationship (e.g., one-to-one) are representing by appropriately shading the diamond. CASE tools are good examples of the effective use of direct manipulation interfaces. A workbench to support the implementation phase for the JSD technique has been developed [2]. The CASE tool has a graphical user interface where a specification can be created, viewed and edited using icons (for the specification) and pull down menus (for the various actions).

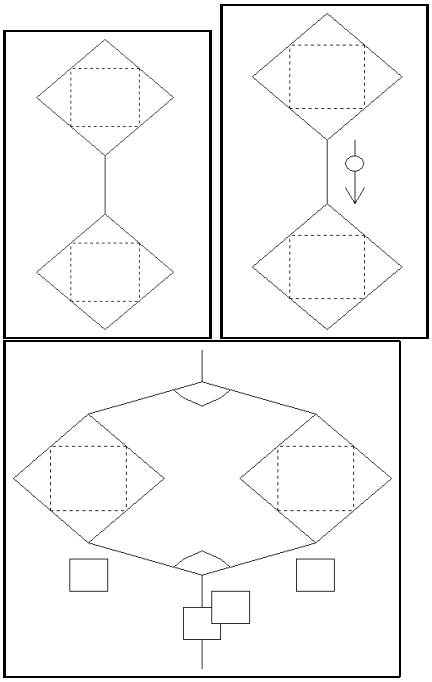
Formal systems have been made easier to use by creating a graphics based front end [16, 8]. The development of a modern user interface to the theorem prover HOL [4] is discussed in [16]. Though they provide a "menu" driven interface, the display and most of the actions are derived directly from the HOL specifications. The user interface is mainly aimed at those who are well versed with HOL. PAM described in [8] is similar in that most of the theory (for concurrent systems) is represented textually with the user interface providing book keeping functions.

In our system, Guile, we also use a menu driven approach, but the actions are represented pictorially. By designing the icon to indicate the appropriate concept, the user of the system is insulated from the burden of manipulating the underlying notational details. While the user must still understand and conceptualise the semantics, the tool aids in the transfer of the cognitive design to the formal notation. Guile can be perceived as a drawing tool where each graphical object represents an action.

Guile has been implemented using the SUIT toolkit [13]. The principal reason for choosing SUIT was that most of

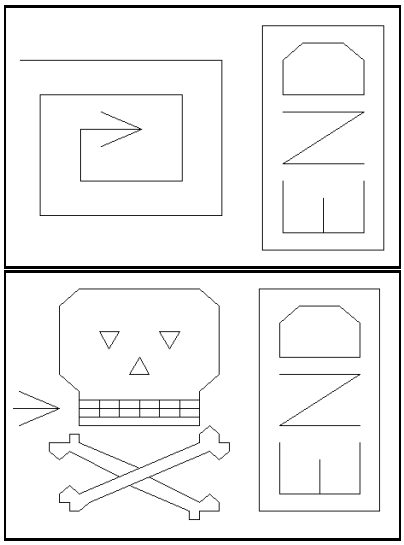
the required widgets (such as pull down menus, scrollable lists) were provided by the toolkit. Although SUIT permitted us to start the implementation of our ideas quickly, we soon encountered some of its limitations of SUIT, especially with the icon drawing aspects. This is not altogether surprising given that the authors in [13] state that they expect users to outgrow SUIT and move onto more complex systems. In section 4 we discuss this issue in more detail.

In the remainder of this section we describe the various features of the tool.



3.1 Action and Combinator Representation

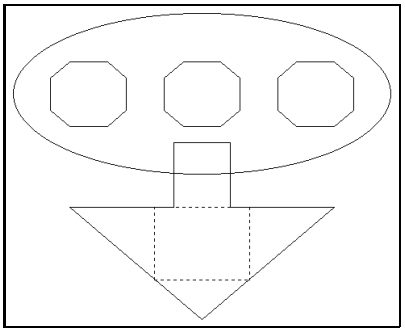
We represent each type of information by a particular geometric shape. Basic actions are represented by their behaviour. For example, the action diverge and fail are represented as shown below.

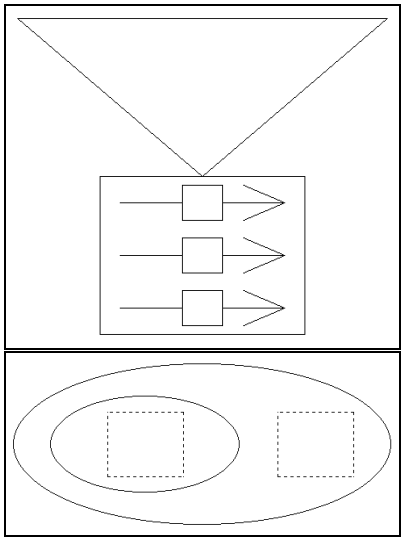


Sub-actions of composite actions are represented by diamonds with the combinator represented by an appropriate connective. For example, the combinators and then, then and moreover are shown below.

The diagrams for and then and then indicates that the combinators are basically sequencing, while the diagram for then shows that the transient value from the first action is given to the second action. The diagram for moreover indicates that the combinator is basically interleaving and the overlapping of the square boxes indicates that the bindings are overlayed with the bindings generated by the second action taking priority. The two diamonds can then be filled with other (perhaps composite) actions.

Similarly, data, bindings, sorts are represented by different symbols. For example, the action choose, the yielder current bindings and the subsort relation are expressed as follows:





The symbols for all the items in a semantic definition is given in Figure 1.

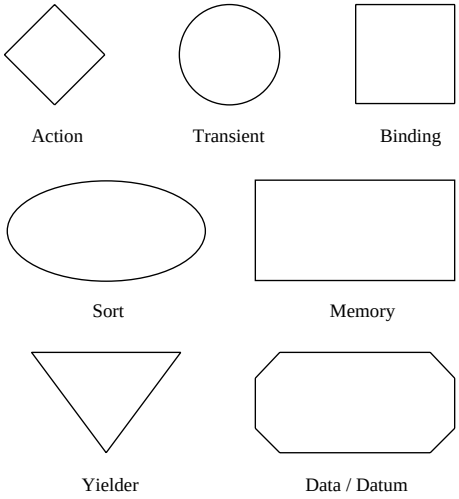


Figure 1: Symbols

In the following sections, we use the example presented in Section 2 to show the various features of the tool.

3.2 Syntax Editor

When Guile is entered the user is presented with a number of menus such as the '**File**' menu shown in Figure 2 which enables language definitions to be '**Load**'ed, '**Save**'ed and displayed using LaTeX.

Selecting '**Syntax**' from the '**Editor**' menu enables the user to define the abstract syntax of the language. Figure 3 shows the situation while the '`Statement`' non-terminal of HypoPL is being defined. New productions of the abstract syntax are introduced by clicking on the '**Production Add**' button. The resulting dialog box

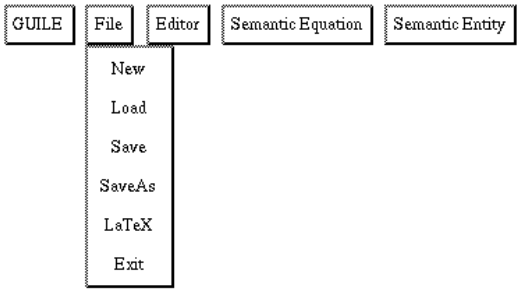


Figure 2: Guile '**File**' menu

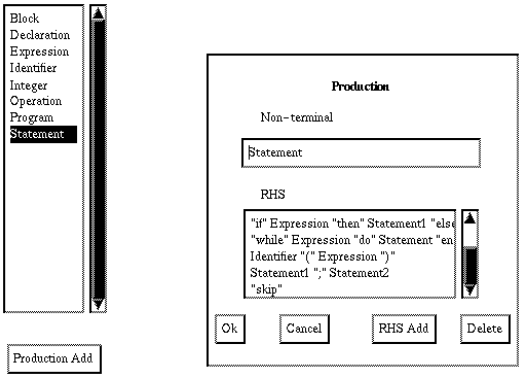


Figure 3: Defining HypoPL abstract syntax with Guile

has an entry for introducing a new non-terminal (in this case '`Statement`') and new productions can be added by clicking of the '**RHS Add**' and completing the resulting dialog box (Figure 4).

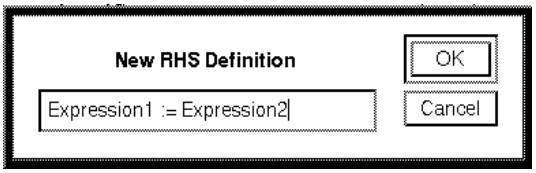


Figure 4: Guile '**RHS Add**' dialog

Some rudimentary checks of the defined grammar are made during entry to avoid common errors such as duplication of definitions. However more complex checks such as checking for the completeness of the grammar has been left to a production version of the tool.

At any stage while the abstract grammar is being defined it is possible to use the Semantic Equation Editor to attach action notation semantics to the productions. The user can switch back and forth between the Syntactic and Semantic editors as required during language definition.

3.3 Semantic Equation Editor

The '**Semantic Equation**' menu (Figure 5) contains entries to

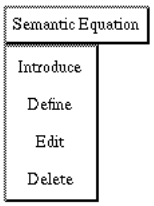


Figure 5: Guile '**Semantic Equation**' menu

'Introduce' new semantic equation function name and attached it to a particular abstract syntax non-terminal,

'Define' a previously introduced semantic equation function by defining the meaning of each production associated with the non-terminal,

'Edit' a previous function definition,

'Delete' an existing semantic equation function name completely.

The user begins by using '**Introduce**' to select which non-terminal is involved and then providing the name of the semantic function (Figure 6). Then for each production of the non-terminal the user '**Define**'s a semantic equation for the function.

For example if the user chooses '**Define**' from the '**Semantic Equation**' menu and then selects 'execute::Statement->action' (Figure 7) they are presented with a scrollbar of those productions of '`Statement`' for which the '`execute`' function has not been defined which at this point would be all productions of '`Statement`' (Figure 8).

Figure 9 shows the situation if the assignment '`Statement`' is selected. The user is presented with a canvas and a palette of icons by means of which the definition can be defined graphically. Any basic action represented by the scrollable palette of icons can be dragged into the main canvas to begin the top-level graphical representation of the semantic equation defining execute⟦ Expression1 ":=" Expression2 ⟧. The icons are separated into a number of separate lists corresponding to their

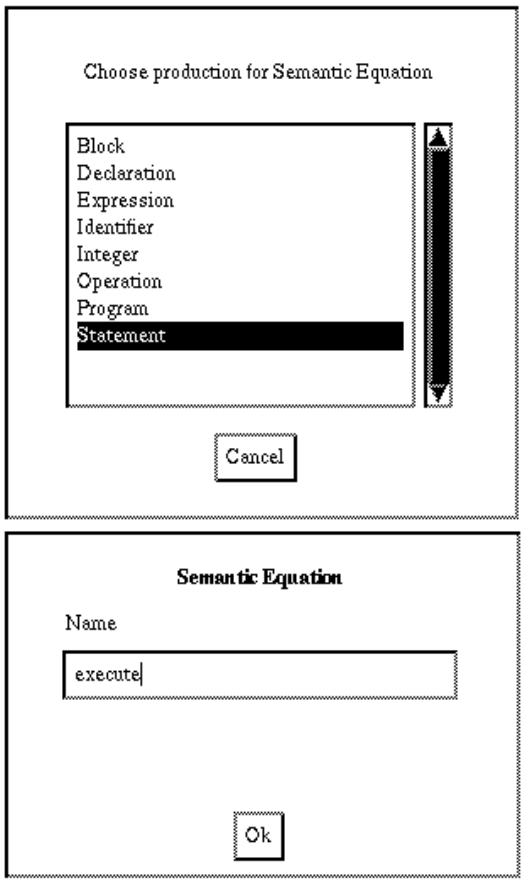


Figure 6: Attaching semantic equation function '`execute`' to '`statement`'

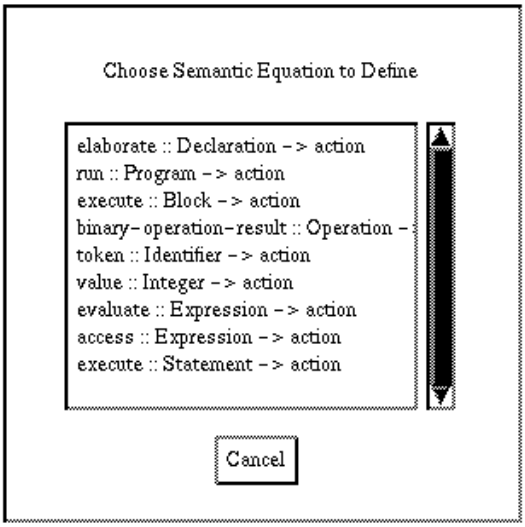


Figure 7: Selecting the newly defined '`execute`' function

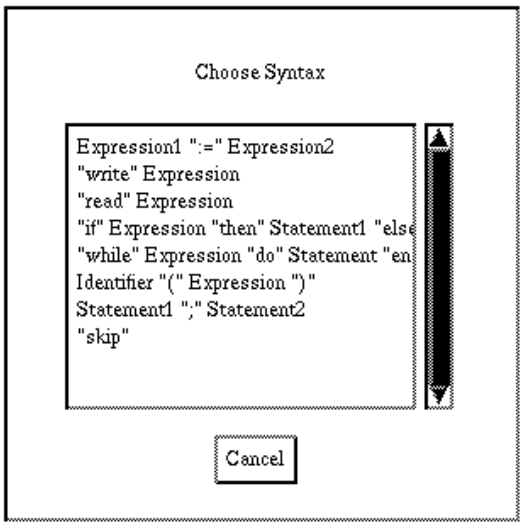


Figure 8: Selecting the assignment statement to define

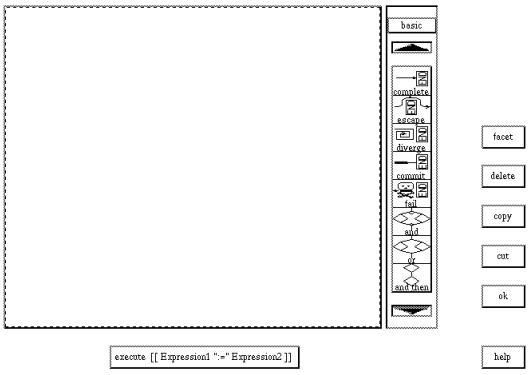


Figure 9: Defining the semantics of '`execute`' function for the assignment '`statement`'

facet ('**basic**', '**functional**', etc.) and the '**facet**' button can be used to switch between these. As explained in Section 3.1 the icons are designed to suggest a particular semantics to the user. In Figure 10 the user has switched to the '**functional**' facet and dragged the '`then`' action icon to the canvas.

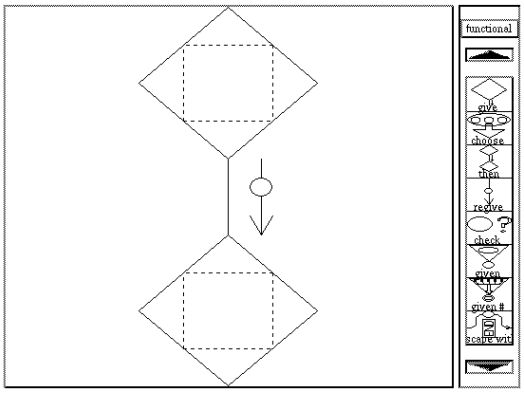


Figure 10: Situation after dragging the '`then`' action icon of the '**functional**' facet to the canvas

The resulting '`then`' diagram represents the (partial) semantic equation execute⟦ Expression1 ":=" Expression2 ⟧ = ___ then ___ with the dotted rectangles indicating that '`then`' requires two sub-diagrams to be defined. It is then possible to drag icons into these two sub-diagrams to define the two parts of the '`then`'. Figure 11 shows the situation after an '`and`' and '`store`' icon has been dragged into the subparts. The size of the icons and the level of detail is reduced for sub-diagrams to make the diagram manageable and as readable as possible. By clicking on a sub-diagram with the left mouse button the user can 'zoom' in on a sub-diagram as in Figure 12 where the user has clicked on the lower sub-diagram of Figure 11. The bottom level diagrams usually require arbitrary text to be included which can be facilitated by clicking on the box with the middle mouse button and then typing the text into the resulting dialog box. By these means the complete definition of the semantic function for the given production can be constructed. The completed definition of the assignment statement is shown in Figure 13.

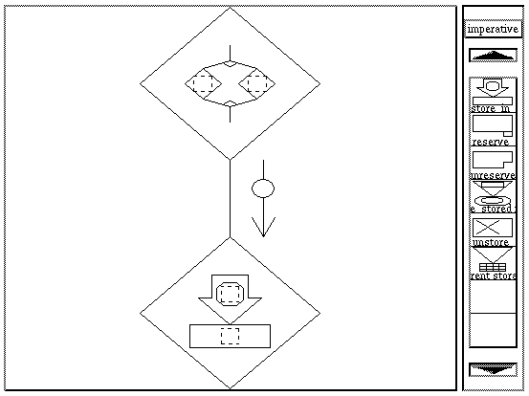


Figure 11: Situation after dragging '**and**' and '**store**' action icons to the canvas

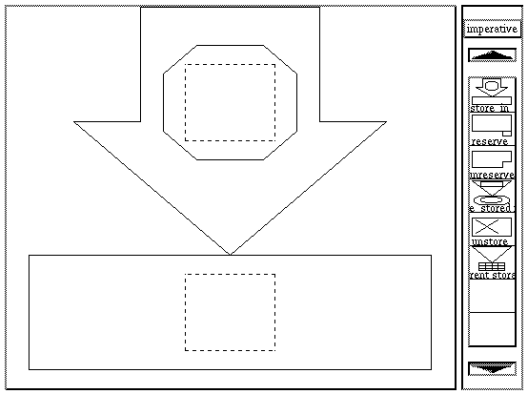


Figure 12: Zooming in on the bottom subdiagram

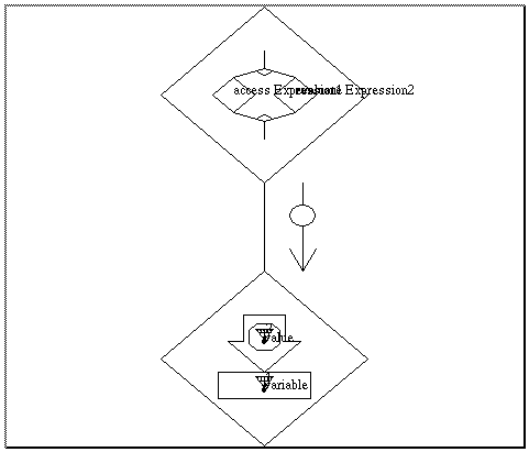


Figure 13: The completed diagram definition of the assignment statement

At a later stage the semantics of the assignment statement could be changed by selecting
'**Edit**' from the '**Semantic Editor**' menu. When the '**execute::Statement->action**' was selected only those '**Statement**' productions that had been defined would be available for selection. Similarly when the '**Define**' menu was used again and '**execute::Statement->action**' selected the assignment statement would not be available indicating a definition had already been provided (Figure 14).

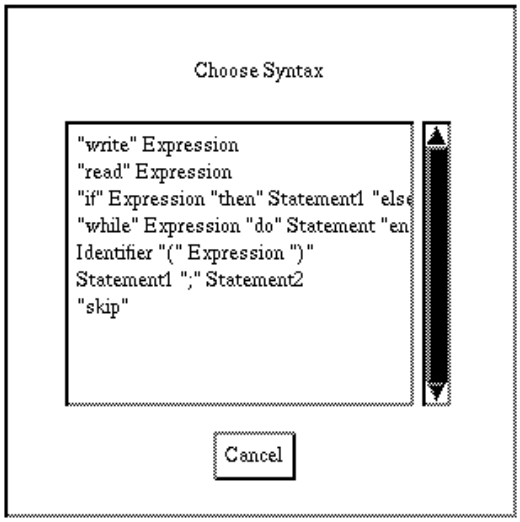


Figure 14: Assignment statement is no longer available for definition

Existing definitions can be edited by means of the '**Edit**' entry under the '**Semantic Equation**' menu. To assist in editing of diagrams rudimentary clipboard features have been provided. The '**copy**' makes a copy of the diagram currently displayed on the canvas and and puts it on a clipboard. The contents of the clipboard are displayed and can be dragged into the main canvas to be used as a component of any diagram.

The '**Semantic Entity**' menu is similar to that of the '**Semantic Equation**' menu and allows functions to be '**Introduce**'d, '**Define**'d, '**Edit**'ed and '**Delete**'d. The only difference is that as these functions are not associated with syntax productions but rather define auxiliary data/actions. The only extra information the user needs to supply is the type of the function. Fig-

ure 15 show the user introducing a 'variable' semantic entity specifying the type of 'variable' as a 'datum'. The definition of this could be the sort union of 'simple-variable', 'complex-variable' and would use the same graphical facilities as used by the Semantic Equation editor. Such a definition is useful to distinguish between values which can stored in a single cell and values (such as arrays/records) which can be stored in a collection of cells.

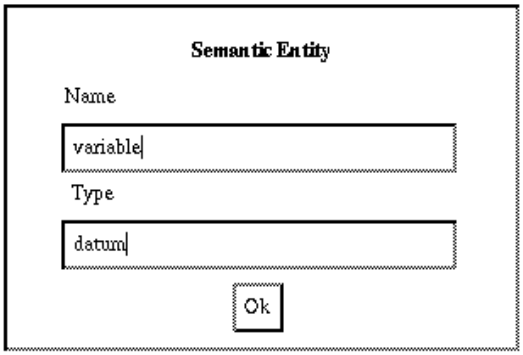


Figure 15: A 'variable' semantic entity is introduced

3.4 Other Extant Features

There is an extensive help facility provided by the '**Help**' button. Help on the meaning of any action icon can be obtained by dragging the icon onto the help button. This results in a dialog box that explains the form and meaning of the icon. (Figure 16)

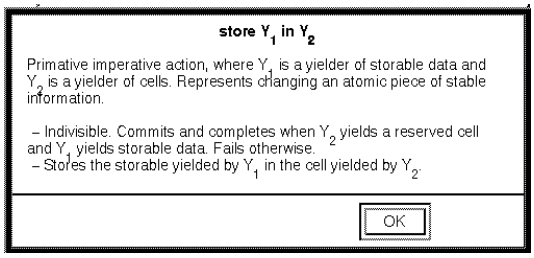


Figure 16: Help obtained by dragging the 'store' icon to the '**Help**' button.

Furthermore if any diagram is copied onto the clipboard it can also be dragged to the help button. This yields the textual action notation for that diagram with any 'holes' in the diagram replaced by 'Undefined'. Figure 17 shows the result of dragging the completed assignment diagram from the clipboard to the '**Help**' button.

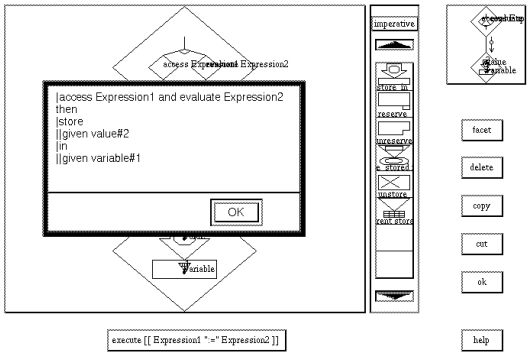


Figure 17: Dragging the assignment diagram to the '**Help**' button after it had been copied to the clipboard.

The LaTeX entry on the '**File**' button produces a well formatted textual representation of the complete language definition. This is achieved by generating a LaTeX file which is then processed with '`latex`' and displayed with '`xdvi`'. Figure 18 presents the LaTeX form of the syntactic category '`Statement`' as shown in Figure 8. Figure 19 presents the LaTeX form of some of the semantic equations.

```
Program     = "program" Identifier Block
Statement   = Expression1 ":=" Expression2 |
              "write" Expression |
              "read" Expression |
              "if" Expression "then" Statement1 "else" Statement2 "endif" |
              "while" Expression "do" Statement "endwhile" |
              Identifier "(" Expression ")" |
              Statement1 ";" Statement2 |
              "skip"
```

Figure 18: LaTeX form of the syntactic definition of statements

4 Conclusions and Future Work

The tool has been used by three people: one experienced in developing action descriptions, one exposed a little to action semantics and one novice.

The novice found the tool useful in both understanding an existing semantics and in developing small descriptions. For example, the diagrams were useful in noticing that differences between some of the

```
run :: Program → act

run ⟦ "program" Identifier Block ⟧ =
        execute Block

execute :: Block → act

execute ⟦ Declaration "begin" Statement "end" ⟧ =

        | furthermore elaborate Declaration
        hence execute Statement

execute ⟦ "begin" Statement "end" ⟧ =
        execute Statement

binary-operation-result :: Operation → act

binary-operation-result ⟦ "+" ⟧ =
        the sum of (the given number #1, the given number #2)

binary-operation-result ⟦ "-" ⟧ =
        the difference of (the given number #1, the given number #2)
```

Figure 19: LaTeX form of some of the semantic equations

'confusing combinators' such as then and and then and and existed. The presence of overlaying bindings and labelled bindings helped distinguish the semantics of moreover and hence.

The intermediate user found the tool helpful to structure the development of the semantics. When trying to decide on the appropriate semantics for a particular language feature, the icons along with the associated help acted as a clue to guide the user towards the correct action. Furthermore the tool released the user from concerns regarding the syntax of the action notation and its formatting. However, some of the user interface features were frustrating to use requiring too many mouse clicks and drags to perform what should be simple operations. For example, using the help facility required dragging the appropriate icon to the help button. A 'hot key' facility can easily fix this problem.

The tool hindered the development of a semantics for an experienced user. The principal reason being that it was easier to develop the semantics in text form (i.e., typing it) than to go through the process of drawing it. However, the written semantics could not necessarily be understood by the novice as the tool does not convert text to diagrams. The tool has not yet been used extensively. More study is necessary before definitive conclusions can be drawn.

The use of SUIT was both an advantage and an disadvantage for this project. The advantages was that standard widgets such as pull down menu's were available directly. The ability to experiment with the various properties, such as size, location, within the prototype interface was useful. The ability to use SUIT as a library for C programs implied that the programmer did not have to be an expert in X windows programming. The principal disadvantage was encountered in implementing the drawing aspects of the tool. When drawing non standard items SUIT requires the program to detect events and handle them. Guile was developed on a Sun SPARCstation SLC where performance was quite good. However it was found that when run on an X-terminal that performance was a problem. The delay between generating an event and SUIT detecting it was significant on a X-terminal. Furthermore redrawing the palette and the canvas was time consuming as it was not done incrementally.

Finally we summarise the work necessary to complete our tool. For Guile to become a programming language designer's workbench, the output of the tool should be processed by a compiler generating system, such as the Actress system [3] or the Cantor system [10, 11]. We are exploring the feasibility of automating this process. More facilities for editing existing definitions need to be implemented. The icons used in the current display are not sufficiently intuitive to all users. An icon editor, which permits the creation of appropriate icons and their use in Guile has been designed and prototyped. This editor needs to the integrated into Guile. A feature to convert a text stream into a diagram would be useful as it would permit the experienced user to create the semantics quickly but also allow the novice to browse through it carefully.

Acknowledgement

Many thanks to Andy Cockburn and the anonymous referees for their helpful comments. This research has been supported by University of Canterbury Grant No 1787123.

References

[1] *Ada programming language (ANSI/MIL-STD-1815A)*. Washington, D.C. 20301, January 1983.

[2] A. Bass, M. Boyle, and B. Ratcliff. PRESTIGE: A CASE Workbench for the JSD Implementor. In *Proceedings of the 13th International Conference on Software Engineering*, pages 198–207, Austin, Texas, 1991.

[3] D. F. Brown, H. Moura, and D. A. Watt. ACTRESS: An Action Semantics Directed Compiler Generator. In U. Kastens and P. Pfahler, editors, *International Workshop on Compiler Construction*, pages 95–109, Paderborn, 1992. Springer-Verlag.

[4] M. Gordon. HOL: A Machine Oriented Formulation of Higher Order Logic. Technical Report 68, University of Cambridge, Computer Laboratory, 1985.

[5] J. Hannan. Making Abstract Machines Less Abstract. In *Functional Programming and Computer Architecture: LNCS 523*, pages 618–635. Springer Verlag, 1991.

[6] J. Hannan. Staging Transformations for Abstract Machines. In *ACM SIGPLAN Symposium on Partial Evaluation and Semantics Based Program Manipulation*, pages 130–141. Sigplan Notices, 1991.

[7] P. Lee. *Realistic Compiler Generation*. The MIT Press, 1989.

[8] H. Lin. Pam: A Process Algebra Manipulator. In K. G. Larsen and A. Skou, editors, *Proc. Third Workshop on Computer Aided Verification: LNCS 575*, pages 136–146, Aalborg, Denmark, 1991. Springer Verlag.

[9] P. D. Mosses. *Action Semantics*. Number 26 in Tracts in Theoretical Computer Science. Cambridge University Press, August 1992.

[10] J. Palsberg. A Provably Correct Compiler Generator. In *European Symposium On Programming (ESOP): LNCS 582*, pages 418–434, Rennes, France, February 1992. Springer-Verlag.

[11] J. Palsberg. An Automatically Generated and Provably Correct Compiler for a Subset of Ada. In *Fourth IEEE International Conference on Computer Languages*, San Fransisco, California, April 1992.

[12] L. Paulson. A Semantics Directed Compiler Generator. In *Ninth ACM Symposium on Principles of Programming Languages*, pages 224–233, January 1982.

[13] R. Pausch, M. Conway, and R.DeLine. Lessons Learned from SUIT, the Simple User Interface Toolkit. *ACM Transactions on Information Systems*, 10(4):320–344, Oct 1992.

[14] U. Pleban. Compiler Prototyping Using Formal Semantics. In *ACM SIGPLAN Symposium on Compiler Construction*, pages 94–105. Sigplan Notices, 1984.

[15] T. J. Teorey, D.Yang, and J. P. Fry. A Logical Design Methodology for Relational Databases Using the Extended Entity-Relationship Model. *ACM Computing Surveys*, 18(2):197–222, June 1986.

[16] L. Thery, Y. Bertot, and G. Kahn. Real Theorem Provers Deserve Real User-Interfaces. In *Proceedings of the 5th Symposium on Software Development Environments*, 1992.